Input Space Partitioning

Dr. Kosta Damevski

CMSC 425/525 - Intro to Software Analysis and Testing

Spring 2023

VCU

Computer Science

College of Engineering

Last Time — Criteria-Based Testing

- *Test Criteria* produce *Test Requirements*
  - e.g. criteria = all lines of code, test requirements = line 1, line 2, line 3…
- Coverage of a test set T is about mapping to the set of Test Requirements
  - Coverage level often less than 100%
- Criteria can subsume one another

Four Structures for Modeling Software for Test

- *Input-Space Partitions*
  - Today!
- Graph
- Logic
- Syntax

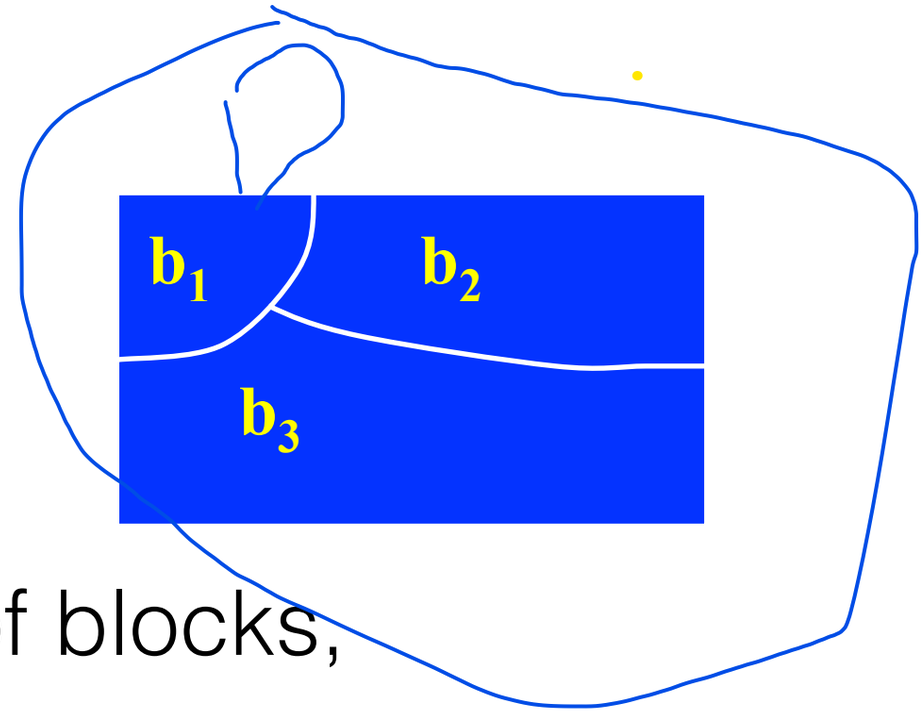
Input Domains

- For even small programs, the input domain is so large that it might as well be infinite
- Testing is fundamentally about *choosing finite sets of values from the input domain*
  - Input parameters define the scope of the input domain
    - Parameters to a method
    - Data read from a file
    - Global variables
    - User level inputs
  - Domains for input parameters are partitioned into *regions*
    - At least one value is chosen from each region

Benefits of Input Space Partitioning

- Can be equally applied at several levels of testing
  - Unit
  - Integration
  - System
- Relatively easy to apply
  - Doesn't require extensive automation
- Easy to adjust the procedure to get more or fewer tests
- No implementation knowledge is needed (i.e. black box)
  - Just the program's input space

Partitioning Domains



- Domain ***D***
- Partition scheme ***q*** of ***D***
- The partition ***q*** defines a set of blocks, ***Bq = b1 , b2 , …, bQ***
- The partition must satisfy two properties :
  - Blocks must be pairwise disjoint (no overlap)
  - Together the blocks cover the domain ***D*** (complete)

Using Partitions

- *Idea: choose a value from each block*
- *Each value (block) is assumed to be equally useful for testing*
- Application to testing:
    1. Find characteristics in the inputs
    2. Partition each characteristic (into blocks)
    3. Choose tests by combining values from characteristics
- Example characteristics
    - Input X is null (yes/no)
    - Order of the input file F (sorted, inverse sorted, arbitrary, …)

Choosing Partitions

- *Choosing (or defining) partitions seems easy, but is easy to get wrong*
- Consider the characteristic "order of file F"
  - block1 = sorted in ascending order
  - block2 = sorted in descending order
  - block3 = arbitrary order
- BUT, what if the file is of length 1
  - The file will be in all three blocks (disjointness is no longer satisfied)
- One possible solution
  - characteristic1: file sorted in ascending order (c1.b1=true; c1.b2=false)
  - characteristic2: file sorted in descending order (c2.b1=true; c2.b2=false)

Choosing Partitions (2)

- If the partitions are not complete or disjoint, that means the partitions have not been considered carefully enough
  - They should be reviewed carefully, like any design
  - Different alternatives should be considered

Complete Set of Steps in Modeling the Input Domain

- Step 1: Identify testable functions
    - e.g. individual methods, methods in a class, external software, hardware, database, etc.
- Step 2: Find all the parameters
    - e.g. all inputs (including various state)
- Step 3: Model the input domain
    - defined in terms of characteristics; one block for each characteristic;
- Step 4: Apply a test criterion to choose combinations of values
    - how to combine blocks and characteristics
- Step 5: Refine combinations of blocks into test inputs
    - choose appropriate values from each block

Two Approaches for Input Domain Modeling

- *Interface-based approach*
  - Develops characteristics directly from individual input parameters int, string, etc
  - Simplest application
  - Can be partially automated in some situations
- *Functionality-based approach*
  - Develops characteristics from a behavioral view of the program under test based on what the program should do
  - Harder to develop—requires more design effort
  - May result in better tests, or fewer tests that are as effective

Interface-Based Approach

- Mechanically consider each parameter in isolation
- This is an easy modeling technique and relies mostly on syntax
- Some domain and semantic information won't be used
- Could lead to an incomplete Input Domain Model (IDM)
- Ignores relationships among parameters

Example of Interface-Based Approach

```
public enum Triangle { Scalene, Isosceles, Equilateral, Invalid }


// side1, side2, and side3 represent the lengths of the sides of a triangle
// Returns the appropriate enum value
public static Triangle triangle (int side1, int side2, int side3)
```

- The input domain model for each parameter is identical
- Reasonable characteristic: *Relation of side to 0*
    - blocks: is_zero; not_zero

Functionality-Based Approach

- Identify characteristics that correspond to the intended functionality
- Requires more design effort from tester
- Can incorporate domain and semantic knowledge
- Can use relationships among parameters
- Modeling can be based on requirements, not implementation
- The same parameter may appear in multiple characteristics, so it's harder to translate values to test cases

Example of Functionality-Based Approach

```
public enum Triangle { Scalene, Isosceles, Equilateral, Invalid }


// side1, side2, and side3 represent the lengths of the sides of a triangle
// Returns the appropriate enum value
public static Triangle triangle (int side1, int side2, int side3)
```

- The three parameters represent a triangle
- Reasonable characteristic: *Triangle type*
    - *blocks*: isosceles, equilateral, scalene, invalid

Combining Interface and Functionality Based IDM

```
// Effects: if list or element is null throw NullPointerException
//          else return true if element is in the list, false otherwise
public boolean findElement (List list, Object element)
```

- *Interface-Based Approach*
  - Characteristics :
    - list is null (block1 = true, block2 = false)
    - list is empty (block1 = true, block2 = false)
- *Functionality-Based Approach*
  - Characteristics :
    - number of occurrences of element in list (0, 1, >1)
    - element occurs first in list (true, false)
    - element occurs last in list (true, false)

Modeling the Input Domain

- Partitioning the input into characteristics and into blocks and values is a creative engineering step
  - More blocks means more tests
- Partitioning often flows directly from the definition of characteristics and both steps are done together
  - Sometimes fewer characteristics can be used with more blocks and vice versa

Interface-Based

Characteristic	b_1	b_2	b_3
q_1 = "Relation of Side 1 to 0"	greater than 0	equal to 0	less than 0
q_2 = "Relation of Side 2 to 0"	greater than 0	equal to 0	less than 0
q_3 = "Relation of Side 3 to 0"	greater than 0	equal to 0	less than 0

- A maximum of 3*3*3 = 27 tests
- Some triangles are valid, some are invalid
- Refining the characterization can lead to more tests

Functionality-Based

- Prior two characterizations are based on syntax–parameters and their type
- A semantic level characterization could use the fact that the three integers represent a triangle

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	isosceles	equilateral	invalid

- Violates "disjoint" property of blocks
  - fix by stipulating b2 block

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	isosceles, not equilateral	equilateral	invalid

Functionality-Based

- Possible values for *Triangle type* characteristic

Characteristic	b_1	b_2	b_3	b_4
Triangle	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

Another Functionality-Based

- A different approach would be to break the triangle types into 4 different characteristic

Characteristic	b_1	b_2
q_1 = "Scalene"	True	False
q_2 = "Isosceles"	True	False
q_3 = "Equilateral"	True	False
q_4 = "Valid"	True	False

Strategies for Identifying Values

- Include valid, invalid and special values
- Sub-partition some blocks
- Explore boundaries of domains
- Include values that represent “normal use”
- Try to balance the number of blocks in each characteristic
- Check for completeness and disjointness

Choosing a Combination of Values

- Once characteristics and partitions are defined, the next step is to choose test values
  - We use criteria – to choose effective subsets
  - The most obvious criterion is to choose all combinations
    - All Combinations of Coverage (ACoC) Criteria

All Combinations of Coverage (ACoC)

- *All combinations of blocks from all characteristics must be used.*
- Number of tests is the product of the number of blocks in each characteristic :
- The second characterization of triang() results in 4*4*4 = 64 tests
    - Too many ?

Input Criteria - All Combinations

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Refinement of q_1"	greater than 1	equal to 1	equal to 0	less than 0
q_2 = "Refinement of q_2"	greater than 1	equal to 1	equal to 0	less than 0
q_3 = "Refinement of q_3"	greater than 1	equal to 1	equal to 0	less than 0

- Let's relabel the blocks for clarity

Characteristic	b_1	b_2	b_3	b_4
A	A1	A2	A3	A4
B	B1	B2	B3	B4
C	C1	C2	C3	C4

Input Criteria - All Combinations

A1 B1 C1	A2 B1 C1	A3 B1 C1	A4 B1 C1
A1 B1 C2	A2 B1 C2	A3 B1 C2	A4 B1 C2
A1 B1 C3	A2 B1 C3	A3 B1 C3	A4 B1 C3
A1 B1 C4	A2 B1 C4	A3 B1 C4	A4 B1 C4
A1 B2 C1	A2 B2 C1	A3 B2 C1	A4 B2 C1
A1 B2 C2	A2 B2 C2	A3 B2 C2	A4 B2 C2
A1 B2 C3	A2 B2 C3	A3 B2 C3	A4 B2 C3
A1 B2 C4	A2 B2 C4	A3 B2 C4	A4 B2 C4
A1 B3 C1	A2 B3 C1	A3 B3 C1	A4 B3 C1
A1 B3 C2	A2 B3 C2	A3 B3 C2	A4 B3 C2
A1 B3 C3	A2 B3 C3	A3 B3 C3	A4 B3 C3
A1 B3 C4	A2 B3 C4	A3 B3 C4	A4 B3 C4
A1 B4 C1	A2 B4 C1	A3 B4 C1	A4 B4 C1
A1 B4 C2	A2 B4 C2	A3 B4 C2	A4 B4 C2
A1 B4 C3	A2 B4 C3	A3 B4 C3	A4 B4 C3
A1 B4 C4	A2 B4 C4	A3 B4 C4	A4 B4 C4

This is almost certainly more than we need

only 8 tests are valid (all sides > 0)

Each Choice Coverage (ECC)

- Another idea: let's try *at least one value from each block*
- Each Choice Coverage (ECC) : One value from each block for each characteristic must be used in at least one test case.
  - Number of tests is the number of blocks in the largest characteristic

A1 B1 C1	(2, 2, 2)
A2 B2 C2	(1, 1, 1)
A3 B3 C3	(0, 0, 0)
A4 B4 C4	(-1, -1, -1)

Pairwise Coverage Criteria (PWC)

- Another idea: select a pair of characteristics and make sure you sample all pairs from them, not caring much about the rest
- Pair-Wise Coverage (PWC) : A value from each block for each characteristic must be combined with a value from every block for each other characteristic.
- Number of tests is at most the product of two largest characteristics

A1, B1, C1	A1, B2, C2	A1, B3, C3	A1, B4, C4
A2, B1, C2	A2, B2, C3	A2, B3, C4	A2, B4, C1
A3, B1, C3	A3, B2, C4	A3, B3, C1	A3, B4, C2
A4, B1, C4	A4, B2, C1	A4, B3, C2	A4, B4, C3

t-Wise Coverage (TWC)

- A natural extension is to require combinations of t values instead of 2
- t-Wise Coverage (TWC) : A value from each block for each group of t characteristics must be combined.

Base Choice Coverage (BCC)

- Testers sometimes recognize that certain values are important
- This uses domain knowledge of the program
- Base Choice Coverage (BCC) : A base choice block is chosen for each characteristic, and a base test is formed by using the base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic

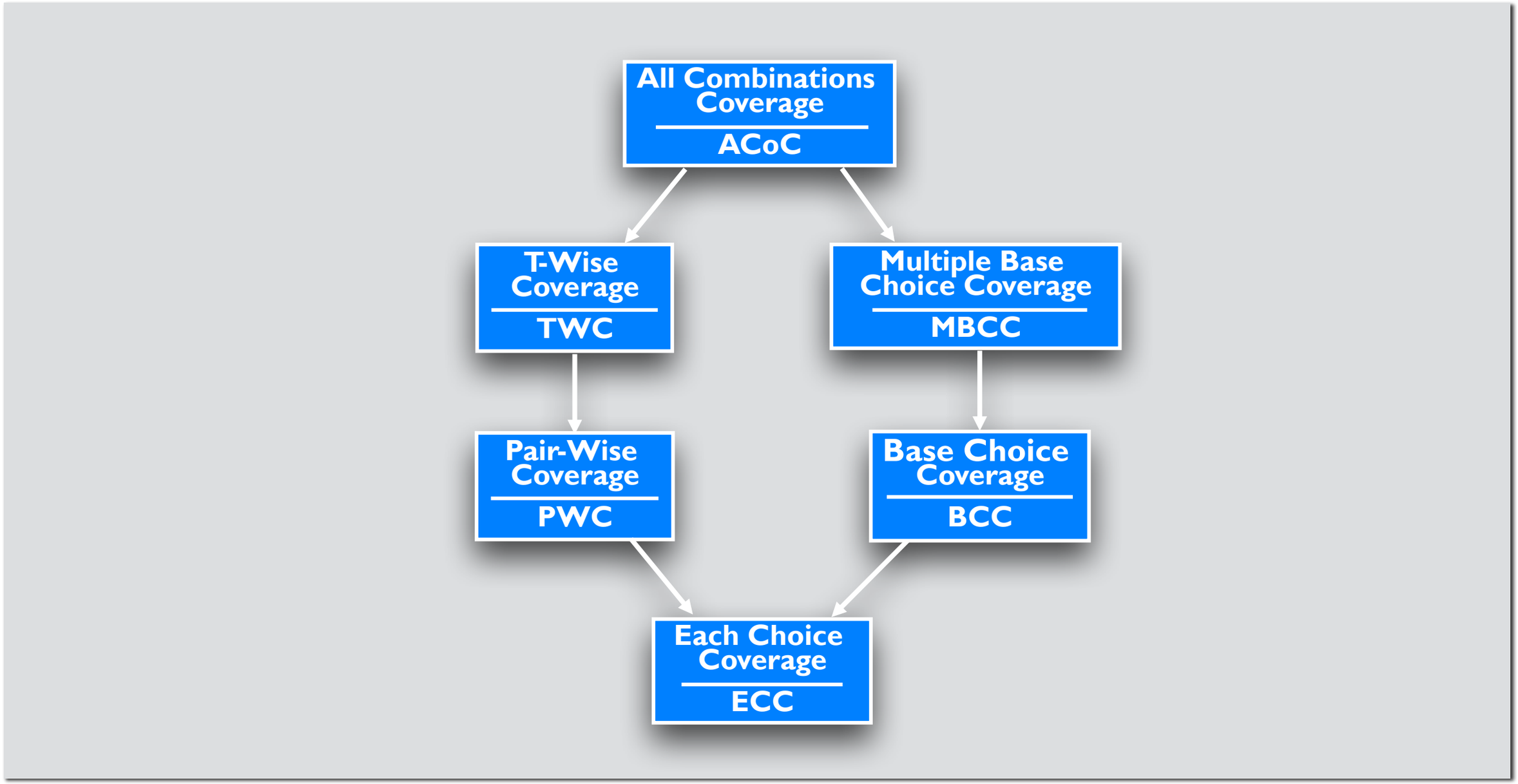
Base A1, B1, C1

A1, B1, C2	A1, B2, C1	A2, B1, C1
A1, B1, C3	A1, B3, C1	A3, B1, C1
A1, B1, C4	A1, B4, C1	A4, B1, C1

Base Choice Coverage

- Base choices can be
  - Most likely from an end-use point of view
  - Simplest
  - Smallest
  - First in some ordering
- Usually should be a happy path test
- The base choice is a crucial design decision
- Extension: Multiple Base Choice Coverage
  - Select more than one base case, then do the same

ISP Coverage Criteria Subsumption



Infeasible Combinations

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          else return true if element is in the list, false otherwise
```

Characteristic	Block 1	Block 2	Block 3	Block 4
A : length and contents	One element	More than one, unsorted	More than one, sorted	More than one, all identical
B : match	element not found	element found once	element found more than once	
Invalid combinations : (**A1**, **B3**), (**A4**, **B2**)				

element cannot be in a one-element list more than once

If the list only has one element, but it appears multiple times, we cannot find it just once

Summary

- Fairly easy to apply, even with no automation
- Convenient ways to add more or less testing
- Applicable to all levels of testing – unit, class, integration, system, etc.
- Based only on the input space of the program, not the implementation (i.e. black box)

References

- “Introduction to Software Testing” 2nd Edition. Ammann and Offutt